

On the Implementation of an Or-Parallel Prolog System for Clusters of Multicores

JOÃO SANTOS and RICARDO ROCHA

CRACS & INESC TEC and Faculty of Sciences, University of Porto

Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal

(e-mail: {jsantos,ricroc}@dcc.fc.up.pt)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Nowadays, clusters of multicores are becoming the norm and, although, many or-parallel Prolog systems have been developed in the past, to the best of our knowledge, none of them was specially designed to explore the combination of shared and distributed memory architectures. In recent work, we have proposed a novel computational model specially designed for such combination which introduces a *layered model* with two scheduling levels, one for workers sharing memory resources, which we named a *team of workers*, and another for teams of workers (not sharing memory resources). In this work, we present a first implementation of such model and for that we revive and extend the YapOr system to exploit or-parallelism between teams of workers. We also propose a new set of built-in predicates that constitute the syntax to interact with an or-parallel engine in our platform. Experimental results show that our implementation is able to increase speedups as we increase the number of workers per team, thus taking advantage of the maximum number of cores in a machine, and to increase speedups as we increase the number of teams, thus taking advantage of adding more computer nodes to a cluster.

KEYWORDS: Or-parallelism, Environment Copying, Scheduling, Implementation, Performance.

1 Introduction

The inherent non-determinism in the way logic programs are structured as simple collections of alternative clauses makes Prolog very attractive for the exploitation of *implicit parallelism*. Prolog offers two major forms of implicit parallelism: *and-parallelism* and *or-parallelism*. And-Parallelism stems from the parallel evaluation of subgoals in a clause, while or-parallelism results from the parallel evaluation of a subgoal call against the clauses that match that call. Arguably, or-parallel systems, such as Aurora (Lusk et al. 1988) and Muse (Ali and Karlsson 1990), have been the most successful parallel Prolog systems so far. Intuitively, the least complexity of or-parallelism makes it more attractive and productive to exploit than and-parallelism, as a first step. However, practice has shown that a main difficulty is how to efficiently represent the *multiple bindings* for the same variable produced by the or-parallel execution of alternative matching clauses. One of the most successful or-parallel models that solves the multiple bindings problem is *environment copying*, which has been efficiently used in the implementation of or-parallel

Prolog systems both on shared memory (Ali and Karlsson 1990; Rocha et al. 1999) and distributed memory (Villaverde et al. 2001; Rocha et al. 2003) architectures.

Another key problem in the implementation of a parallel system is the design of *scheduling strategies* to efficiently assign tasks to workers. In particular, with implicit parallelism, it is expected that the parallel system automatically identifies opportunities for transforming parts of the computation into concurrent tasks of parallel work, guaranteeing the necessary synchronization when accessing shared data. For environment copying, scheduling strategies based on *dynamic scheduling of work* using *or-frame data structures* to implement such synchronization have proved to be very efficient for shared memory architectures (Ali and Karlsson 1990). *Stack splitting* (Gupta and Pontelli 1999; Pontelli et al. 2006) is an alternative scheduling strategy for environment copying that provides a simple and clean method to accomplish work splitting among workers in which the available work is *statically divided beforehand* in complementary sets between the sharing workers. Due to its static nature, stack splitting was first introduced aiming at distributed memory architectures (Villaverde et al. 2001) but, recent work, also showed good results for shared memory architectures (Vieira et al. 2012).

Nowadays, the increasing availability and popularity of multicores and clusters of multicores provides an excellent opportunity to turn Prolog an important member of the general ecosystem of parallel computing environments. However, although many parallel Prolog systems have been developed in the past (Gupta et al. 2001), most of them are no longer available, maintained or supported. Moreover, to the best of our knowledge, none of those systems was specially designed to explore the combination of shared and distributed memory architectures. In recent work, we have proposed a novel computational model (Santos and Rocha 2014) which addresses the problem of efficiently exploit the combination of shared and distributed memory architectures. Such proposal introduces a *layered model* with two scheduling levels, one for workers sharing memory resources, which we named a *team of workers*, and another for teams of workers (not sharing memory resources), that somehow resembles the concept of teams used by some models combining and-parallelism with or-parallelism, like the Andorra-I (Santos Costa et al. 1991) or ACE (Gupta et al. 1994) systems, where a layered approach also implements different schedulers to deal with each level of parallelism.

In this work, we present a first implementation of such proposal and for that we revive and extend the YapOr system (Rocha et al. 1999) to efficiently exploit parallelism between teams of workers running on top of clusters of multicores. YapOr is an or-parallel engine based on the environment copying model that extends the Yap Prolog system (Santos Costa et al. 2012) to exploit implicit or-parallelism in shared memory architectures. Our platform takes full advantage of Yap’s state-of-the-art fast and optimized engine and reuses the underlying execution environment, scheduler and part of the data structures used to support parallelism in YapOr. In our previous work (Santos and Rocha 2014), we have described the high-level algorithms that support the key aspects of the layered model. In this paper, we focus the description on the operational aspects of our specific implementation of the model, such as: how a parallel engine is created; how a parallel goal is launched; how the team scheduler is structured in different modules; how we deal with load balancing and termination; how we have implemented the sharing work process using an auxiliary sharing area; etc.

In order to take advantage of our platform, we also propose a new set of built-in pred-

icates that constitute the syntax to interact with an or-parallel engine in our platform. Experimental results show that our implementation is able to increase speedups as we increase the number of workers per team, thus taking advantage of the maximum number of cores in a machine, and to increase speedups as we increase the number of teams, thus taking advantage of adding more computer nodes to a cluster.

The remainder of the paper is organized as follows. First, we briefly introduce the key aspects of the layered model. Next, we present the new syntax and execution model of our platform and discuss the most important implementation details. At the end, we present experimental results and outline some conclusions and further work directions.

2 Layered Model

The goal behind the layered model is to implement the concept of *teams of workers* and specify a clean interface for *scheduling work between teams* (Santos and Rocha 2014). A team is defined as a set of workers (processes or threads) who share the same memory address space and cooperate to solve a certain part of the main problem. At the team level, or-parallelism can thus be explored by using any of the available strategies tailored for scheduling/distributing work among workers in shared memory.

To schedule/distribute work between teams, the layered model introduces a second-level team-based scheduler based on the general ideas of the environment copying model with stack splitting. The second-level scheduler specifies a clean and common interface for scheduling work between teams, which fully avoids shared data structures for synchronization by following a static work splitting strategy among teams.

Figure 1 shows a schematic representation of a possible scenario for an or-parallel engine *E1* running on top of a cluster formed by two host computer nodes. Host *N1* has two teams, team *A* and team *B*, with 3 and 4 workers each, and host *N2* has one team, team *C*, with 8 workers. The strategy adopted to distribute work

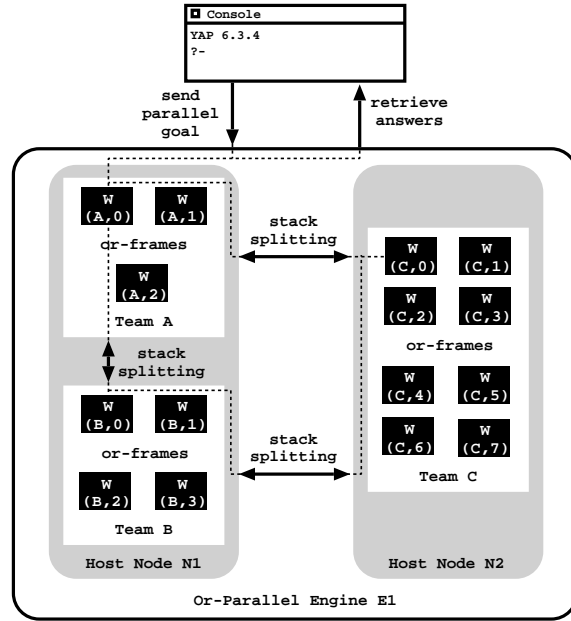


Fig. 1. Schematic representation of the layered model

inside each team is based on dynamic scheduling with or-frames, while among teams is based on static scheduling with stack splitting. This idea is similar to the MPI/OpenMP hybrid programming pattern, where MPI is usually used to communicate among different computer nodes and OpenMP is used to communicate among workers in the same node. In what follows, we focus our description on the operational aspects of our specific implementation of the layered model.

3 Our Platform

3.1 Syntax

Our proposal for the set of built-in predicates that constitute the syntax to interact with an or-parallel engine follows two important design rules: (i) avoid blocking mechanisms for interacting with an or-parallel engine, and (ii) delegate to the programmer the responsibility of *explicitly annotate* which parts of the program should be run in an or-parallel engine in order to exploit implicit or-parallelism. Our proposal includes five predicates:

par_create_parallel_engine(+EngName, +ListOfTeams): creates and launches the teams and workers that form a new or-parallel engine. The first argument defines the name to be given to the new or-parallel engine and the second argument is a list of tuples defining the teams to be created. Each tuple $team(h, n, p)$ includes the host computer node h where the team should be launched, the number of n workers to be spawned on that team and the path p to the file program to be loaded by default. For example, the following call could be used to create the topology illustrated in Fig. 1:

```
par_create_parallel_engine('E1', [team('N1',3,'prog.pl'), team('N1',4,'prog.pl'),
                                  team('N2',8,'prog.pl')]).
```

par_run_goal(+EngName, +Goal, Template): asynchronously runs a goal in an or-parallel engine. It receives as arguments the name of the or-parallel engine where to run the goal, the goal to be run and a template indicating how answers should be returned. Consider, for example, that we want to run the goal $start(X, Y)$ in engine $E1$ and we are only interested in the answers obtained for Y . In such case, we should call:

```
par_run_goal('E1', start(X, Y), Y).
```

par_probe_answers(+EngName): checks if the or-parallel engine given as argument has found new answers for the current parallel goal or has already finished its execution. If so, it succeeds. Otherwise, it fails. Using the current example would be:

```
par_probe_answers('E1').
```

par_get_answers(+EngName, +Mode, -ListOfAnswers, -NumOfAnswers): asynchronously retrieves answers from an or-parallel engine. It receives as arguments the name of the or-parallel engine, how many answers to try to retrieve, the list where to return the answers found and the number of answers returned. This predicate is not backtrackable, however when called again it will return a new set of answers (if new answers exist). When all answers have been retrieved and the parallel goal has finished, it simply fails. The available options for the second argument are:

- **max(N)** – attempts to retrieve up to N answers, meaning that it will not block if the number of available answers is less than N .
- **exact(N)** – attempts to retrieve exactly N answers, meaning that it will block while the number of answers is less than N or the execution has not finished.

For example, if we want to retrieve exactly 10 answers from engine $E1$, we could write:

```
par_get_answers('E1', exact(10), ListOfAnswers, NumOfAnswers).
```

par_free_parallel_engine(+EngName): terminates all the workers and teams for the or-parallel engine given as argument. Using again the current example would be:

par_free_parallel_engine('E1').

3.2 Execution Model

At the beginning of the execution only one standard Yap engine, called the *client worker*, is running. As expected, the client worker is responsible for interacting with the user and execute the user's queries. In order to allow the parallel execution of goals, it is necessary to create beforehand, at least, one or-parallel engine (our platform allows to create several independent or-parallel engines and run different parallel goals on each engine).

The worker 0 of each team is named the *master worker* of the team and it is responsible for launching the execution inside the team and for the communication with the other teams. Moreover, the first team to be launched is named the *master team* and its *master worker* is also responsible for launching the execution of the parallel goals sent by the client worker and for returning the found answers. As described before, Fig. 1 shows an example of an or-parallel engine, where we can see all these communication links (dotted lines) between teams and with the client worker (Yap's console on top of the figure).

If during the execution of a user's query, the client worker calls a *par_run_goal/3* predicate, then a message with the goal to be run in parallel is sent from the client worker to the master worker of the master team, worker $W(A,0)$ in Fig. 1. The client worker then continues its computation, either executing other sub-computations, checking for the existence of answers for the parallel goal, or retrieving such answers when available. At the same time, worker $W(A,0)$ will notify all the other master workers about the new received parallel goal and then start its execution. Inside the master team, the execution behaves like an independent YapOr engine, with the master worker sharing work with its teammates. Outside the master team, the other teams are now aware that a parallel computation has begun and thus they enter in *team scheduling mode*. Soon after, they will start contacting the master team in order to get work. When a team A receives a sharing work request from a team B , the *team sharing work process* begins.

In the team sharing work process, first a worker W from team A is chosen to answer the request. Then, worker W may reject or accept the request based on its current conditions. If the request is accepted, W proceeds as follows. Initially, worker W starts by copying its stacks to an auxiliary area assign to itself. Then, a stack splitting strategy is applied to its stacks and the stacks in the auxiliary area. Finally, the stacks in the auxiliary area are sent to the master worker of the requesting team B . In the continuation, when the master worker of team B receives the stacks from team A , they are installed on its own workspace. At this point, the master worker of team B informs the remaining teammates that the team has now work. After that, the execution inside team B evolves as a standard YapOr execution with the master worker performing a fail, in order to take the next open alternative, and with its idle teammates starting to ask it for work.

A team is considered to be out of work when every worker inside the team is idle. When that happens, the team enters again in team scheduling mode in order to choose a busy team to request work and the same sharing work process, as described above, is repeated. The execution ends when all teams are idle and, in the continuation, the client worker is notified that the parallel execution is finished.

Our platform can be seen as a layer implemented on top of Yap which extends the existing YapOr’s shared memory approach to support work sharing between independent YapOr engines. For both layers we follow an approach based on environment copying techniques. Our choice is because environment copy showed to be one of the most successful models and because YapOr already implements it as the basis for the team concept. A team can be thought as a YapOr engine using or-frames to synchronize the access to the open alternatives inside the team. On top of that, a second layer responsible for distributing work among teams was developed from scratch. This second layer supports two different stack splitting techniques to distribute work among teams and the communication between teams is implemented with MPI messages.

4 Implementation Details

4.1 Creating a Parallel Engine and Starting a Parallel Goal

Predicate *par_create_parallel_engine/2* initiates the process of creating an or-parallel engine. Initially, it uses the *MPI_Comm_spawn_multiple()* function of the MPI API to spawn the master workers of each team. Then, a set of MPI messages inform each master worker about the number of workers to be present in the corresponding team. In the continuation, each master worker allocates the shared memory which will support the parallel execution of its team and launches the remaining workers of the team using the *fork()* system call, in a process similar to the one in YapOr (Rocha et al. 1999). Each worker will then initialize its environment and, at last, jump to the *getwork_first_time* instruction. At that point, the parallel engine is ready to run.

Algorithm 1 shows the pseudo-code for the *getwork_first_time* instruction. As we can see, all master workers start by waiting for their teammates to become ready and, after that, they guarantee that all teams are in the same condition (i.e., all master workers wait for each other). The first wait condition is implemented in shared memory by marking the ready workers in a bitmap while the second wait is implemented using a MPI barrier.

After this initial synchronization, the master worker of the master team waits for a new parallel goal to be sent by the client worker (line 5 in Alg. 1). Remember that predicate *par_run_goal/3* triggers such synchronization. When that happens, the goal and template strings received are converted to Prolog terms and then the parallel computation starts by running a *parallel/2* predicate with the goal and the template terms as arguments.

The remaining master workers wait for a signal (line 10 in Alg. 1) from the master worker of the master team to be sent by the *run_engine()* procedure (line 8 in Alg. 1). That signal is done in a form of a message that contains information such as the memory position of the root choice point, that will allow all master workers to allocate their own root choice points in the same memory position (this is important for the environment copying technique). In the continuation, each master worker executes the *team_idle_scheduler()* procedure which starts the process of looking for a busy team to request work to (line 12 in Alg. 1).

On the other hand, all (non-master) workers wait for a notification from their master worker saying that the team has work. After receiving such notification, they also allocate their own root choice points (line 16 in Alg. 1), this time such information can be read directly, through shared memory, from the workspace of the master worker, and then

Algorithm 1 *getwork_first_time(worker W , team T)*

```

1: if is_master_worker( $W$ ) then
2:   wait_for_tammates()
3:   wait_for_master_workers()
4:   if is_master_team( $T$ ) then
5:      $msg = wait\_for\_message\_from\_client()$ 
6:      $goalTerm = get\_goal(msg)$ 
7:      $templateTerm = get\_template(msg)$ 
8:      $run\_engine(parallel(goalTerm, templateTerm))$ 
9:   else {non-master team}
10:    wait_for_work_in_engine()
11:    allocate_root_choice_point()
12:    team_idle_scheduler()
13:  else {non-master worker}
14:    set_worker_as_ready( $W$ )
15:    wait_for_work_in_team()
16:    allocate_root_choice_point()
17:    local_scheduler()

```

they enter in scheduling mode in order to search for work inside the team (line 17 in Alg. 1). Every time a team runs out of work, the non-master workers return to this instruction waiting again for its master worker to get work from another team. From their perspective, every time the master work gets work from the outside, is like beginning a new parallel computation.

4.2 Team Scheduler

The team scheduler can be divided in two different modules. The first module, called *team idle scheduler*, runs when all workers in a team are out of work. Once it happens, the master worker of the team enters in scheduling mode and starts running the team idle scheduler in order to find a busy team willing to share work. The second module, called *team busy scheduler*, is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams. Together, both modules of the team scheduler implement several functionalities, such as: (i) the handling of the communications between teams; (ii) the sharing work process between teams; and (iii) the termination process, which determines the end of the execution of a parallel goal by ensuring that all search space was fully explored.

In order to better understand how the team scheduler works, let us consider the example in Fig. 2. On the left side of the figure, we have an idle team A formed by three workers and, on the right side we have a busy team B formed by four workers. The boxes inside both teams show the main components of the two modules of the team scheduler. The arrows represent information exchanged during scheduler execution – if they are exchange between workers of the same team we call them notifications (dotted arrows in Fig. 2), otherwise, if they correspond to information exchange between workers from different teams we call them messages (solid arrows in Fig. 2).

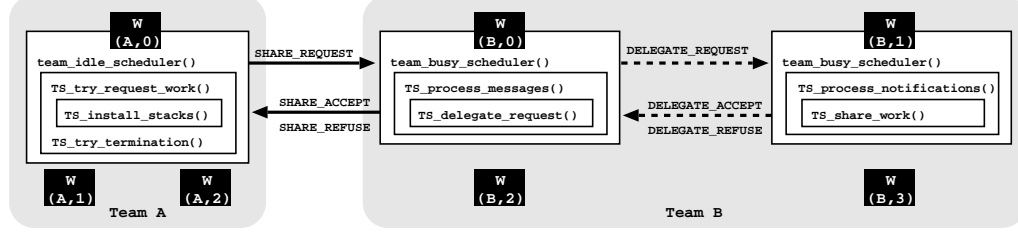


Fig. 2. Main components of the team scheduler

Since team A is idle, the master worker $W(A,0)$ starts running the team idle scheduler and, in particular, the $TS_try_request_work()$ procedure responsible for selecting a busy team to request work to. In the example, the procedure's algorithm decided to send a sharing request to the busy team B . When the master worker $W(B,0)$ in team B notices that there are pending messages coming from other teams, it runs the $TS_process_messages()$ procedure. The request from team A will then call the $TS_delegate_request()$ procedure in order to select the worker inside the team with the best conditions to successfully answer the request. In this example, the selected worker was worker $W(B,1)$ and therefore a $DELEGATE_REQUEST$ notification is sent to it. Even though the sharing request involves two teams, in practice, the sharing process is done between two workers. The master worker of the requesting team and the chosen sharing worker from the busy team.

Worker $W(B,1)$ also runs the team busy scheduler from time to time looking for delegation requests and once it receives one, the $TS_process_notifications()$ procedure is called. If the request is accepted, $W(B,1)$ then calls the procedure $TS_share_work()$ that is responsible for applying the stack splitting technique, preparing the stacks to be shared and returning a $DELEGATE_ACCEPT$ notification to the master worker $W(B,0)$. Otherwise, a $DELEGATE_REFUSE$ notification informs $W(B,0)$ that the request was rejected. When the master worker $W(B,0)$ receives the $DELEGATE_ACCEPT$ notification, it returns a $SHARE_ACCEPT$ message together with the shared stacks to the requesting team A and, the master worker $W(A,0)$ in team A , runs the $TS_install_stacks()$ procedure to install the received stacks in its workspace. At this point, team A is no longer considered to be idle and the computation returns to Prolog execution. On the other hand, if the sharing request is denied with a $SHARE_REFUSE$ message, team A would try to initiate the termination process by calling the $TS_try_termination()$ procedure.

4.3 Load Balancing and Termination

An important goal of our team scheduler is to achieve an efficient distribution of work between teams in order to optimize resource usage and thus minimize response time. A good strategy, when searching for a team to request work, would be to select the busy team that holds the highest *work load*, i.e., the highest amount of unexplored alternatives. Nevertheless, selecting such a team would require having precise information about the work load of all teams, which might not be possible without introducing a considerable communication and synchronization overhead. A more reasonable solution is to find a compromise between the load balancing efficiency and the implementation overhead.

In our implementation, each worker holds a *load register*, as a measure of the exact

number of open alternatives in its private choice points, and each team holds a *load array*, as a measure of the estimated work load of each fellow team. The load register is used to support the process of selecting the worker with the highest load inside the team, either when a teammate runs out of work or when delegating a sharing request. The load array is used to support the process of selecting a busy team to request work.

The team's load information is stored in a bi-dimensional array which contains, for each team, its work load and a timestamp. We define the load of a team as the sum of the particular loads of each worker in the team. The load array is updated whenever a team message is received. We do not introduce specific messages to explicitly ask for work load information and, instead, the existing messages are extended to include that information. When a master worker sends a new message, it includes a copy of its load array, so that the receiving master worker can update its load array with such information. This is done by comparing the timestamps in both arrays. When a received timestamp is younger than the current timestamp for a given team, then the load array is updated with the received value. Timestamps are implemented using an integer which is incremented every time the corresponding team sends a new message.

From a conceptual point of view, the load array can be seen as the view that a team has about the other teams in the or-parallel engine. Therefore, it can be useful not only for selecting teams with work but also for detecting termination, i.e., when no busy team exists. However, in an extreme scenario, we may have a team with load 0 but that is still busy. This may happen if, for example, all open alternatives are in the shared region of the team. In order to distinguish this situation, we represent the load of a team out of work as -1, instead of 0. The termination process is thus triggered when all teams have a load value of -1 in the load array. This is a safe conclusion because of the way our timestamp mechanism is used to update the load array in each team, which ensures that if a team reaches a situation where all load values are -1 for all teams, then the computation as ended. In such case, a termination message is sent to all other teams signaling that the computation has ended. Otherwise, we restart the process of requesting work by sending sharing requests to the workers with higher load or load 0. Meanwhile, as other teams refuse sharing work, they will send their load array which may contain newer information that could help to decide if the computation has ended or not.

4.4 Delegated Sharing Process

When a worker receives a delegated sharing request, its stacks may not be sent directly to the requesting team because only MPI processes can send messages and, in our implementation, the non-master workers are non-MPI processes since they are launched using the *fork()* system call. The *TS_delegate_request()* procedure thus starts by assigning an *auxiliary sharing area* to each particular delegated request. This is done by initializing a *delegation frame data structure* associated with the request, which allows to implement the communication between the master worker and the selected sharing worker.

The *TS_share_work()* procedure is then responsible for preparing the stacks to be sent to the requesting team. It starts by determining the stacks segments to be copied. In order to better understand how this is done, observe the left side of Fig. 3, which presents the stacks in a worker area. The area of the heap to be copied is delimited by the pointer to the heap in the root choice point, which represents the first shared choice point, and

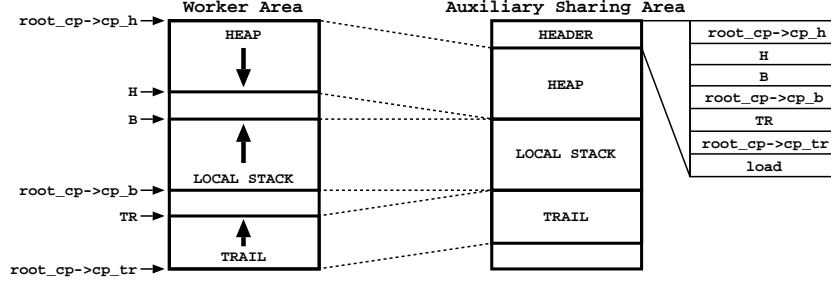


Fig. 3. Preparing the stacks to be sent to the requesting team

the register H, which points to the top of that stack. For the local stack, the area to be copied is delimited by register B, that points to the last choice point, and by the root choice point. For the trail, the area to be copied is given by register TR, that points to the top of the trail, and by the pointer to the trail in the root choice point.

The values that delimit those areas are stored in the header region of the auxiliary sharing area together with the load of the new team that will be determined during the stack splitting operation. Following the header region are the heap, local stack and trail segments as determined before. Since the information in the auxiliary sharing area will be sent to the requesting team, we try to reduce the size of that message by copying those segments in such a way that there is no gaps between them. A schematic view of these steps is depicted on the right side of Fig. 3.

Now we have two copies of the stacks, one in the workspace of the sharing worker and another in the auxiliary sharing area, so we can perform the stack splitting operation between both. After the stack splitting operation completes, the load of the sharing worker and the load in the header are both updated to reflect the changes done and the auxiliary sharing area is ready to be sent to the requesting team. Once the master worker of the requesting team receives it, it just needs to install the stacks in its own workspace with the help of the information present in the header.

4.5 Stack Splitting

Our platform implements two alternative stack splitting strategies for sharing work between teams, namely *vertical splitting* and *horizontal splitting* (Gupta and Pontelli 1999). Figure 4 shows a schematic representation of the vertical stack splitting operation.

In Fig. 4(a) we can see the execution tree of the sharing worker which, initially, is the same as the stacks in the auxiliary area, as explained above. The execution tree of a worker is divided into a *shared* and *private* region. In the shared region, a worker has the nodes (choice points) that are shared with some of its teammates (nodes associated with or-frames in the figure). On the private region, a worker has its private (non-shared) nodes. In Fig. 4(a), the top node is dead, i.e., without open alternatives, but the second and third public nodes have two (*b2* and *b3*) and one (*c3*) open alternatives, respectively. The two private nodes have two more open alternatives each (*d2*, *d3*, *e2* and *e3*).

To implement vertical splitting, the sharing worker should alternately divide its non-dead nodes between its execution stacks and the stacks in the auxiliary sharing area. In Fig. 4(b), we can see the representation of the execution tree in the sharing worker and in

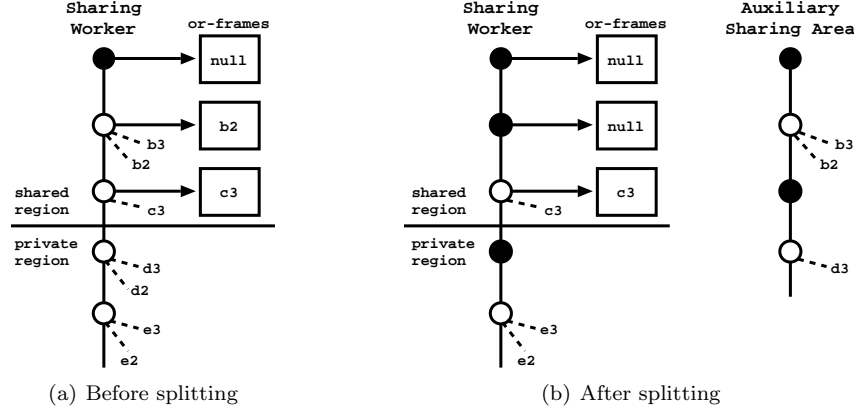


Fig. 4. Vertical stack splitting

the auxiliary sharing area after the vertical stack splitting operation. It is important to note that all nodes, including the ones in the shared region, are considered for splitting. When splitting work in a public node, first the sharing worker needs to gain (lock) access to the corresponding or-frame, copy the next open alternative from the or-frame to the corresponding node in the auxiliary area, update the or-frame to *null* and unlock it. In the auxiliary sharing area, the or-frames associated to the shared choice points can be simply ignored because, after stack splitting, the work is divided in complementary sets and such work will be installed from scratch in another team.

In the case of the horizontal splitting strategy, instead of splitting the nodes, the sharing worker splits the open alternatives in each node. In order to implement this strategy, a new field called *split offset* was added to the choice point and or-frame data structures. This new field allows to calculate the alternatives belonging to each team after a sharing process. It is initialized with a value of one when a node is created and its value is doubled each time the node is split with another team. When a node is turned public, the value in the *split offset* of the choice point is simply copied to the field with the same name in the corresponding or-frame. With the *split offset*, the difference is that, when backtracking, instead of trying the next alternative, as usual, we use the *split offset* field to calculate that alternative. For example, if the *split offset* is two, then instead of trying the next alternative *alt*, we jump *alt* and we try the alternative after *alt* (offset of two).

5 Performance Evaluation

The environment for our experiments included two parallel machines, each one with four AMD SixCore Opteron TM 8425 HE @ 2.1 GHz (24 cores per machine, 48 cores in total) and 64 GBytes of main memory each, both running Fedora 20 with the Linux kernel 3.19.8-100 64 bits. The two machines are connected through a one Gbit router shared with other servers. Our platform was implemented using Yap 6.3.4 and OpenMPI 1.7.3.

Since we had only available two parallel machines with the same characteristics, we tried to emulate the existence of more computer nodes, such that, we could create scenarios where each team of workers always runs on a separate computer node. In order to do that, we acted in the following way: (i) we configured OpenMPI to use the loopback

interface and the TCP protocol for communications between processes even if they are in the same machine (by default OpenMPI uses shared memory for this type of communications); (ii) we used the *tc* command to add more 0.06 milliseconds to the 0.02 milliseconds of latency in the loopback interface in order to simulate the latency that we have observed between the two physical machines, which is about 0.08 milliseconds.

We next present a sequence of experiments using a set of 10 well-known benchmark programs¹. We tried to use benchmark programs widely used in the evaluation of similar or-parallel models, but adapted to take longer when they were too small. All together, the 10 benchmarks take around 1800 seconds (30 minutes) to run with YapOr with a single worker.

Table 1 shows the speedup results achieved for 16, 24 and 32 workers with different configurations of teams when compared with the execution of YapOr with a single worker. We have not collected results for more than 16 workers per machine (32 workers in total) because we do not had full access to the machines and since other users could be using the machines simultaneously, thus interfering with our results, we decided to be safer to go only until 32 workers. All teams are created with the same number of workers and divided equitably between the two physical machines. For example, consider the case of 24 workers and 4 teams, then we launch two teams in each machine and each team is created with six workers each.

The results presented in Table 1 are the average of 10 runs. The table is divided in three parts: execution with 16, 24 and 32 workers. The columns represent the configurations of teams tested where the number of teams in which workers are divided increases from left to right. For example, configuration **[8,8]** means 2 teams with 8 workers each and configuration **[4,4,4,4]** means 4 teams with 4 workers each. For each configuration, we show the results for both vertical and horizontal splitting, respectively, columns **VS** and **HS**. The only exception is the case of 16 workers where we show the results for a single YapOr engine (configuration **[16]**). For this configuration, in parentheses, we also show the execution times in milliseconds for YapOr with a single worker, which were used as the base times to compute the speedups for all configurations in Table 1.

Please note that most of the configurations in Table 1 were not possible without our platform. Before our work, parallelism could only be explored using either shared memory based models or distributed memory based models. With shared memory based models, we are limited by the number of cores in the shared memory architecture. With distributed memory based models, we can easily increase the number of available cores, but we are limited by the potential communication and synchronization costs as we increase the number of workers. For example, consider the scenario where we have two multicore machines with 16 cores each (32 cores in total). With shared memory based models, we can only take advantage of the maximum number of cores in one machine (i.e., configuration **[16]** in Table 1). With distributed memory based models, we can exploit the 32 available cores but the potential benefits of having shared memory resources in the multicore machines are mostly wasted. In Table 1, the configurations **[1,1,...,1]** with a single worker per team match the distributed memory based model implementing the original stack splitting approach (Gupta and Pontelli 1999). We can thus see the results obtained

¹ Our platform and benchmark programs are available at <https://github.com/jpbsantos/yapor-teams>.

Table 1. Speedup results against YapOr execution with a single worker for different configurations of teams with 16, 24 and 32 workers using vertical (VS) and horizontal (HS) splitting on a set of 10 well-known benchmark programs

16 Workers		[16]		[8,8]		[4,4,4,4]		[2,2,...,2]		[1,1,...,1]	
Program	YapOr	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
<i>arithmetic_puzzle</i>	3.39 (361.439)	6.10	6.08	8.07	8.04	9.91	10.20	8.31	7.79		
<i>cubes</i>	15.87 (67.503)	14.29	13.31	12.94	12.65	9.82	10.43	5.31	6.45		
<i>ham</i>	16.01 (121.444)	14.17	14.15	13.49	13.74	6.50	11.60	6.92	6.39		
<i>knight_move(13)</i>	15.70 (395.602)	15.36	14.94	14.86	14.73	13.33	13.30	10.51	10.24		
<i>magic_cube</i>	15.78 (46.872)	12.03	13.95	11.26	12.73	9.00	11.77	4.76	7.32		
<i>map_colouring</i>	15.81 (178.729)	15.17	14.72	14.54	14.06	12.23	12.15	7.11	7.17		
<i>nsort(12)</i>	15.89 (406.292)	14.24	15.48	13.91	14.76	12.98	13.75	10.23	9.91		
<i>puzzle4x4</i>	15.76 (17.538)	11.61	12.56	10.02	9.94	7.36	7.62	3.36	4.56		
<i>queens(14)</i>	16.09 (552.275)	15.00	15.46	14.33	15.03	11.88	14.10	8.09	11.23		
<i>send_more</i>	15.75 (69.684)	14.02	13.97	13.02	13.59	9.87	12.60	5.82	8.36		
Average	14.60	13.20	13.46	12.64	12.93	10.29	11.75	7.04	7.94		

24 Workers		[12,12]		[6,6,6,6]		[4,4,...,4]		[2,2,...,2]		[1,1,...,1]	
Program	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS	
<i>arithmetic_puzzle</i>	6.65	6.56	10.52	10.41	12.02	12.10	11.69	11.72	8.33	8.36	
<i>cubes</i>	20.36	19.09	18.61	17.61	16.66	16.59	11.35	12.00	5.54	6.83	
<i>ham</i>	20.83	20.49	19.81	20.06	16.86	18.25	6.66	14.33	7.01	6.47	
<i>knight_move(13)</i>	22.92	22.06	22.07	21.54	21.12	20.49	17.59	17.02	12.68	11.50	
<i>magic_cube</i>	17.26	20.51	16.20	18.09	14.52	17.04	9.43	13.54	4.11	7.34	
<i>map_colouring</i>	22.39	20.91	20.79	20.29	20.11	18.58	14.07	13.88	7.44	7.29	
<i>nsort(12)</i>	21.02	22.42	20.28	21.58	19.42	20.82	16.78	17.92	11.52	9.88	
<i>puzzle4x4</i>	15.34	16.85	13.00	13.78	11.34	11.83	7.71	8.18	3.49	4.76	
<i>queens(14)</i>	21.97	23.09	21.30	22.48	20.07	21.83	13.71	20.06	9.95	12.74	
<i>send_more</i>	19.67	18.84	19.04	18.97	16.91	18.05	12.51	14.84	5.79	8.52	
Average	18.84	19.08	18.16	18.48	16.90	17.56	12.15	14.35	7.59	8.37	

32 Workers		[16,16]		[8,8,8,8]		[4,4,...,4]		[2,2,...,2]		[1,1,...,1]	
Program	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS	
<i>arithmetic_puzzle</i>	6.70	6.63	11.07	10.61	15.47	15.14	13.45	12.69	8.94	8.79	
<i>cubes</i>	26.04	22.99	23.59	21.27	19.27	19.65	11.72	12.63	5.61	7.27	
<i>ham</i>	26.18	26.65	25.31	26.28	20.13	21.52	6.91	15.36	7.52	7.42	
<i>magic_cube</i>	21.65	26.03	20.28	22.92	15.94	20.81	10.02	14.20	4.57	7.85	
<i>knight_move(13)</i>	30.08	28.10	28.97	28.16	26.81	26.16	20.30	18.09	13.39	12.27	
<i>map_colouring</i>	29.00	25.75	27.46	26.14	23.26	21.96	14.72	14.29	7.88	7.16	
<i>nsort(12)</i>	27.80	28.38	26.50	27.23	24.07	25.05	17.56	18.03	11.55	11.24	
<i>puzzle4x4</i>	19.95	21.21	15.25	17.73	13.77	13.21	7.77	8.54	3.40	5.05	
<i>queens(14)</i>	28.67	30.62	27.55	29.42	23.99	27.99	15.59	23.64	8.74	13.29	
<i>send_more</i>	24.91	22.99	23.86	23.74	19.93	22.55	12.80	15.98	5.45	8.23	
Average	24.10	23.94	22.98	23.35	20.26	21.40	13.08	15.35	7.70	8.86	

for configurations $[1,1,...,1]$ as a way to measure and compare the former distributed memory based models with the newer teams' configurations.

Analyzing the general picture of Table 1, one can observe that regarding the strategy used to distribute work among teams, our experiments seem to suggest that, on average, horizontal splitting achieves slightly better speedup results than vertical splitting and that the difference between both seems to increase as we increase the number of teams (thus also increasing the number of potential stack splitting operations).

Reading the results in Table 1 horizontally (i.e., considering a fixed total number of workers), the speedup results clearly increase as we increase the number of workers per team (thus reducing the number of teams). The exception is the *arithmetic_puzzle* program which seems to benefit from the higher number of stack splitting operations when we have more teams. In this regard, the worst results are for configurations $[1,1,\dots,1]$, which clearly shows the potential of our approach of grouping workers in teams.

Reading the results in Table 1 vertically (i.e., considering a fixed number of workers per team), the speedup results clearly increase as we increase the number of teams. This mimics the scenario where we add more computer nodes with the same characteristics to our cluster. It is interesting to note that the speedup results obtained for configurations $[1,1,\dots,1]$ are, on average, very close in the three sets of workers, 7.04, 7.59 and 7.70 using vertical splitting and 7.94, 8.37 and 8.86 using horizontal splitting. This clearly shows the limitations of the original stack splitting approach when dealing with more workers not grouped in teams. On the other hand, our platform scales better for the configurations with more workers per team.

6 Conclusions and Further Work

We have presented a first implementation of an or-parallel Prolog system specially designed to explore the combination of shared and distributed memory architectures in clusters of multicores and we have proposed a new set of built-in predicates that constitute the syntax to interact with an or-parallel engine in our platform.

Experimental results showed that our platform is able to increase speedups as we increase the number of workers per team, thus taking advantage of the maximum number of cores in a machine, and to increase speedups as we increase the number of teams, thus taking advantage of adding more computer nodes to a cluster. We thus argue that our platform is an efficient and viable alternative for exploiting implicit or-parallelism in the currently available clusters of low cost multicore architectures.

Despite these good results, our platform still suffers from limitations that could be a basis to further research in the area. Examples include: (i) support incremental copying between teams; (ii) avoid speculative work; (iii) support dynamic code compilation; and (iv) support dynamic teams. Further work also includes the ability to pass specific information to the execution system about the parallel computations at hand by using pre-defined directives. We may have directives to define the execution model and the scheduling strategy to be used, directives to define the number of workers, directives to define granularity and load balancing policies, or directives to define restrictions regarding speculative work and predicate side-effects. We also plan to integrate our approach with ThOr (Santos Costa et al. 2010), a thread-based extension of YapOr which uses threads instead of processes to implement implicit or-parallelism, and with another extension of YapOr which implements stack splitting on shared memory (Vieira et al. 2012), thus allowing for the possibility of using stack splitting to share work inside teams.

Acknowledgments

This work was funded by the ERDF (European Regional Development Fund) through the COMPETE 2020 Programme and by National Funds through FCT (Portuguese Founda-

tion for Science and Technology) as part of projects UID/EEA/50014/2013 and ELVEN (POCI-01-0145-FEDER-016844) and through the NORTE 2020 (North Portugal Regional Operational Programme) under the PORTUGAL 2020 Partnership Agreement as part of project NanoSTIMA (NORTE-01-0145-FEDER-000016). João Santos was funded by the FCT grant SFRH/BD/76307/2011.

References

- ALI, K. AND KARLSSON, R. 1990. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming* 19, 2, 129–162.
- GUPTA, G. AND PONTELLI, E. 1999. Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *International Conference on Logic Programming*. The MIT Press, 290–304.
- GUPTA, G., PONTELLI, E., ALI, K., CARLSSON, M., AND HERMENEGILDO, M. V. 2001. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems* 23, 4, 472–602.
- GUPTA, G., PONTELLI, E., HERMENEGILDO, M. V., AND SANTOS COSTA, V. 1994. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*. The MIT Press, 93–109.
- LUSK, E., BUTLER, R., DISZ, T., OLSON, R., OVERBEEK, R., STEVENS, R., WARREN, D. H. D., CALDERWOOD, A., SZEREDI, P., HARIDI, S., BRAND, P., CARLSSON, M., CIEPIELEWSKI, A., AND HAUSMAN, B. 1988. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*. Institute for New Generation Computer Technology, 819–830.
- PONTELLI, E., VILLAVERDE, K., GUO, H.-F., AND GUPTA, G. 2006. Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. *Journal of Parallel and Distributed Computing* 66, 10, 1267–1293.
- ROCHA, R., SILVA, F., AND MARTINS, R. 2003. YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In *Portuguese Conference on Artificial Intelligence*. Number 2902 in LNAI. Springer, 136–150.
- ROCHA, R., SILVA, F., AND SANTOS COSTA, V. 1999. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Portuguese Conference on Artificial Intelligence*. Number 1695 in LNAI. Springer, 178–192.
- SANTOS, J. AND ROCHA, R. 2014. A Team-Based Scheduling Model for Interfacing Or-Parallel Prolog Engines. *Journal of Computer Science and Information Systems* 11, 4, 1435–1454.
- SANTOS COSTA, V., DUTRA, I., AND ROCHA, R. 2010. Threads and Or-Parallelism Unified. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue* 10, 4–6, 417–432.
- SANTOS COSTA, V., ROCHA, R., AND DAMAS, L. 2012. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming* 12, 1 & 2, 5–34.
- SANTOS COSTA, V., WARREN, D. H. D., AND YANG, R. 1991. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 83–93.
- VIEIRA, R., ROCHA, R., AND SILVA, F. 2012. Or-Parallel Prolog Execution on Multicores Based on Stack Splitting. In *International Workshop on Declarative Aspects and Applications of Multicore Programming*. ACM Digital Library.
- VILLAVERDE, K., PONTELLI, E., GUO, H., AND GUPTA, G. 2001. PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In *International Conference on Logic Programming*. Number 2237 in LNCS. Springer, 27–42.